KAST: K-Associative Sector Translation for NAND Flash Memory in Real-Time Systems

Hyunjin Cho
School of ICE
Sungkyunkwan University
Suwon, Korea
hjcho@ece.skku.ac.kr

Dongkun Shin
School of ICE
Sungkyunkwan University
Suwon, Korea
dongkun@skku.edu

Young Ik Eom
School of ICE
Sungkyunkwan University
Suwon, Korea
yieom@ece.skku.ac.kr

***Abstract*—Flash memory is a good candidate for the storage device in real-time systems due to its non-fluctuating performance, low power consumption and high shock resistance. However, the garbage collection for invalid pages in flash memory can invoke a long blocking time. Moreover, the worst-case blocking time is significantly long compared to the best-case blocking time under the current flash management techniques. In this paper, we propose a novel flash translation layer (FTL), called KAST, where user can configure the maximum log block associativity to control the worst-case blocking time. Performance evaluation using simulations shows that the overall performance of KAST is better than the current FTL schemes as well as KAST guarantees the longest block time is shorter than the specified value.**

I. Introduction

Flash memory has been widely used as a storage device for mobile embedded systems (such as MP3 players, PDAs and digital cameras) because of its low-power consumption, non-volatility, high random access performance and high mobility. Over the past several years, there has been a significant growth in the NAND flash market due to the increase of MP3 player and digital camera since these devices need a large amount of data storage. Recently, solid-state disk (SSD), which is made of NAND flash memory, is substituted for hard disk in general purpose computing such as desktop PC [4], [1].

Since flash memory has a high shock resistance, it is useful for real-time systems such as industrial control systems, automobile and space shuttle which require a high reliability. In addition, flash memory does not require a fluctuating seek time unlike hard disk, so it is favorable to providing a predictable I/O performance to the real-time systems.

However, flash memory has also a fluctuation on its bandwidth compared to DRAM due to its special features. This is a critical obstacle to using flash memory in real-time systems. First one is its "erase-before-write" architecture. To write a data in a block, the block should be erased before. Second feature is the unit sizes of erase operation and write operation are different. While the flash memory is erased by the unit of block, the write operation is performed by the unit of page. A block is a bundle of several pages. For example, in Samsung's large block NAND flash memory, one block is composed of 64 pages and the size of a page is 2KB.

Due to these features, special software called flash translation layer (FTL) is required, which provides a logical (virtual) address space to the file systems and maps the logical address to the physical address in flash memory devices. There are two kinds of mapping schemes depending on the mapping granularity, block-level mapping and page-level mapping. Block-level mapping permits only the block-level update. Therefore, even when only one page of a logical block is updated, all the pages of the block should be rewritten at the newly-allocated physical block. So, it requires a high management cost.

For a fast performance, the page-level mapping technique is popular. For example, if an application writes a data to the logical page address 100, the page-level mapping writes the data to the physical page address 200 and records the mapping information at its mapping table. If the application sends an update request on the data at the logical page address 100, FTL writes the new data at the physical page address 201, invalidates the data at the physical page address 200 and updates the mapping information for the logical page address 100. This is because we cannot overwrite the flash memory page.

After a large number of page updates on flash memory, there will be many invalid pages. So, a garbage collector is required to reclaim the space occupied by invalid pages. The garbage collector should scan all the flash memory blocks which have both valid pages and invalid pages, copy valid pages into other clean blocks and erase the blocks. This is a very time-consuming operation, thus invokes a fluctuation on the response time of flash memory. Though Chang et al. [2] proposed a real-time garbage collector for flash memory which uses the page-level mapping, the page-level mapping requires too large mapping table to be used for a large-sized flash memory such as SSD.

So, the hybrid-level mapping is popular recently. In this scheme, all the physical blocks are separated between log blocks and data blocks. The log blocks are called log buffer. The FTL using hybrid-level mapping scheme is called as a log buffer-based FTL. While the log blocks use the page-level mapping scheme, the data blocks are handled by the block-level mapping. The maximum number of log blocks is given to limit the size of page-level mapping table. When a write request is sent to FTL, the data is first written into a log block and the corresponding old data in data block is invalidated. When the log blocks are full and there is no empty space, one

978-3-9810801-5-5/DATE09 © 2009 EDAA

log block is selected as a victim and all the valid pages in the log block are moved into the data blocks to make a log block for on-going write requests. This step is called *block merge*. Since the block merge is performed for only one log block, the block merge cost is smaller than the garbage collection in the page-level mapping. So, the log buffer-based FTL is suitable to real-time systems. In addition, it requires a smaller-sized mapping table than the page-level mapping.

However, depending on the log block association policy to be explained at section 2, the block merge operation of current log buffer-based FTL schemes could impose a long blocking time on real-time tasks at worst-case scenario. For a time-critical system, it is highly important to reduce the worst-case latency so as to higher the system utilization. Although researchers have proposed various hybrid mapping FTLs focusing on the total I/O cost [5], [7], [3], [8], [6], there is little work done in providing a non-fluctuating performance for flash-memory storage systems.

This paper is motivated by the needs of FTL scheme to provide non-fluctuating performance to real-time systems. We propose a novel configurable log buffer management scheme, called KAST, which enables users to control the worst-case blocking time of FTL. In addition, it provides a better I/O performance than the previous FTL schemes.

The rest of the paper is organized as follows. In section 2, the related works are introduced. section 3 describes the overall architecture and the details of KAST scheme. Experimental results are presented in section 4. section 5 concludes with a summary.

II. Related Works

There have been many researches on log buffer-based FTL. The log buffer-based FTLs are divided into three kinds depending on the block association policy as shown in Figure 1, i.e., block associative mapping (BAST) [5], fully associative mapping (FAST) [7] and set associative mapping (SAST) [8]. The block association policy means how many data blocks are associated with a log block. In the BAST (Block Associative Sector Translation) scheme, a log block is allocated for only one data block. In the FAST (Fully Associative Sector Translation) scheme, a log block can have update informations of any data blocks. In the SAST (Set Associative Sector Translation) scheme, a set of log blocks can have pages of a set of data blocks. The distinction among these three schemes is similar to that of three cache architectures (direct-mapped, fully-associative, and set-associative).

In Figure 1(a), there are 6 data blocks and 4 log blocks. We assume that each block has four pages. When updates on data blocks are required by write requests, the log buffer-based FTL writes the new data into the log blocks invalidating the pages in the data block. (The invalidated pages have gray color.) The log blocks with the logical log block number (LLBN) L0, L1, L2 and L3 have the pages for data blocks with the logical block number (LBN) B0, B1, B4 and B5, respectively. For example, since the data block B4 has the page with the logical page number (LPN) p16, the log block L0 is associated

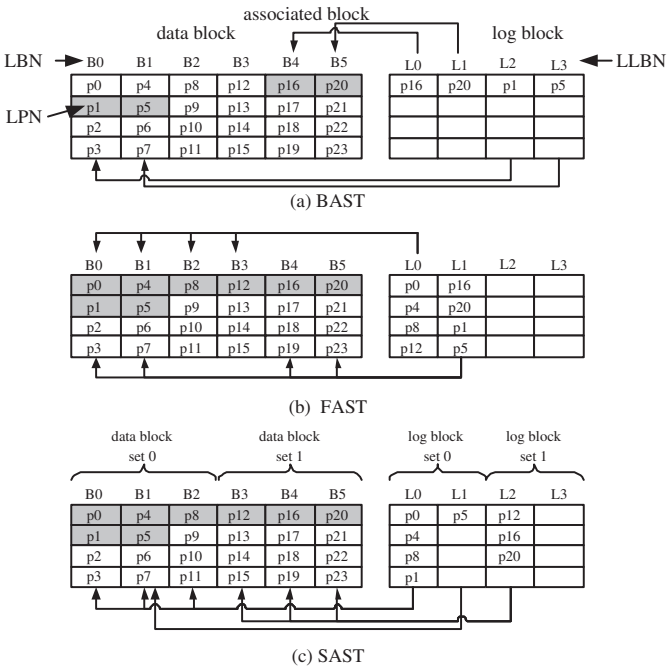


Fig. 1. Log buffer-based FTL schemes.

to B4. If a write request on the data blocks B2 or B3 arrives, one of log blocks should be merged with its associated data block and a new log block should be assigned for the write request. There are three kinds of merges, full merge, partial merge and switch merge [4]. The partial merge and switch merge can be possible only when all the pages within the log block are written sequentially at the page offset determined by their logical page address which is called the in-place scheme.

If the write request pattern is random, the BAST mapping scheme shows poor performance since frequent log block merges are inevitable. This is called the *log block thrashing* problem [7]. For instance, assume that the write pattern is the sequence of "p0, p4, p8, p12, p16, p20, p1, p5". For every write, starting from the write on p16, BAST should replace one of four log blocks generating an expensive merge operation. Figure 1(a) shows the final status of flash memory. Moreover, every victim log block in this example holds only one page when it is replaced; the other three pages remain empty. So, the log blocks in BAST would show very low space utilization when they are replaced from the log buffer.

To solve the problem of BAST scheme, FAST scheme was introduced. In FAST scheme, a log block can be used for any data blocks at a time as shown in Figure 1(b), thus reducing the number of block merges. The pages are written into a log block in the order of requests regardless of the corresponding data block number. So, we can prevent the log block thrashing problem. For the write sequence of "p0, p4, p8, p12, p16, p20, p1, p5", there is no block merge in FAST while each page write incurs a block merge in BAST. As a special case, FAST uses one sequential log block (SLB). A SLB is associated to only one data block. All the pages in the SLB are written by

the in-place scheme. So, the SLB can be merged by either switch merge or partial merge.

However, the problem of FAST scheme is its high block associativity. For example, when the log block L0 in Figure 1(b) is merged with its associated data blocks, 16 number of page copies (16 pages = 4 blocks $\times$ 4 pages) are required since the log block L0 is associated with four data blocks, B0, B1, B2 and B3. This means that FAST scheme incurs a large merge cost. We denote the set of data blocks which are associated to the log block L as $A(L)$. The number of elements in $A(L)$ is called as the block associativity of L and denoted as $k(L)$. Then, the maximum merge cost for a log block L is as follows [6]:

$$N \cdot k(L) \cdot C_{copy} + (k(L) + 1) \cdot C_{erase}$$

where C_{copy} and C_{erase} are the costs of page copy and block erase, respectively. N is the number of pages in a block.

Since the maximum value of $k(L)$ is the number of pages in a block, the cost of $N^2C_{copy}+(N+1)C_{erase}$ are needed in the worst case under the FAST scheme. For the MLC (Multi-Level Cell) NAND flash memory, where a block consists of 128 pages, and the times for page read, page program and block erase are 50μs, 650μs and 2ms, respectively, the worst-case time of a log block merge is about 6 seconds while the best-case time is 2ms in the case of switch merge. So, the worst-case cost is 3,000 times of the best-case cost.

Consequently, BAST scheme shows a poor performance due to its log block thrashing problem and FAST scheme shows a significant difference between the worst-case block merge time and the best-case block merge time due to its high block associativity. Therefore, neither of them is suitable to the real-time systems. In this paper, we propose a novel FTL scheme which uses K-associative mapping similar to FAST but can control the worst-case log block merge time by adjusting the value of K.

In the SAST scheme [3], [8], N log block set can be used only for one data block set which consists of K number of data blocks (N:K log block mapping). Therefore, it is guaranteed that the maximum block associativity of a log block is K. Figure 1(c) shows an example of the 2:3 mapping SAST scheme. Since the SAST scheme is a compromised version of BAST scheme and FAST scheme, it cannot solve both the block thrashing problem and the high block associativity problem completely. For example, it invokes a log block thrashing when the corresponding data block set of write requests are frequently changed.

III. K-Associative Mapping

A. Overall Architecture

The KAST scheme has following special features:

- To minimize the associativities of log blocks, the write requests on the different LBNs are distributed among different log blocks.
- Each log block is enforced to be associated to only the K number of data blocks at maximum. So, we can guarantee that $k(L) \leq K$. By adjusting the value of K, the worst-case log block merge time is controlled.
- There are multiple SLBs unlike the FAST scheme which has only one SLB. However, there is the maximum number of SLBs. The SLB can be changed into a random log block (RLB) if there is no subsequent sequential write request.

While SAST makes the block associativity of log block to be lower than K by associating only K number of data blocks to a log block set, KAST does not group data block and log block. By grouping data block and log block, SAST may meet the block thrashing problem inevitably. KAST only limits the maximum value of associativity of each log block. In addition, KAST tries to minimize the value of $k(L)$ of each log block L by distributing write requests on different LBNs among different log blocks.

Figure 2 shows an example of log buffer usage in the KAST scheme. For the write sequence of "p0, p4, p8, p12, p16, p20, p1, p5", KAST distributes write requests for different data blocks. First, the pages p0, p4, p8, and p12 are written to different log blocks. Then, the pages p16 and p20 are written at the log blocks L0 and L1 increasing the block associativities of the log blocks. Lastly, the pages p1 and p5 are written at the log block L0 and L1 in order not to increase the associativities of log blocks.

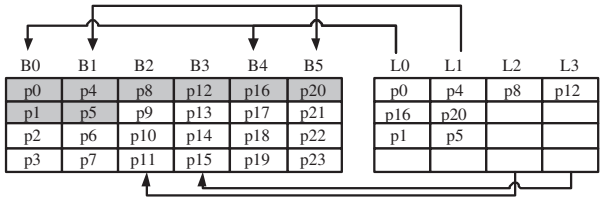


Fig. 2. K-Associative Log Block Mapping.

As shown in Figure 2, KAST tries to minimize $k(L)$ as long as possible. Only when there is no free log block and no allocated log block for the corresponding LBN, KAST increases the block associativity of a log block. As a special case, the SLB is associated to only one data block. When a log block is allocated, it is determined to be an SLB or an RLB.

Figure 3 shows the state transition diagram of a log block. The states F, S and R_i mean the free block, the sequential log block and the random log block with the block associativity i, respectively. Initially, all log blocks are free block without any data (F state). The free block is changed to an SLB (S state) or an RLB (R_1 state). As long as KAST writes a page p to the RLB L with R_i state where $d(p) \in A(L)$ ($d(p)$ denotes the corresponding data block of the page p), the log block L sustains its state R_i. If KAST appends a page to an SLB without breaking the sequentiality of the SLB, the SLB sustains its state S. If a page p is written into an RLB L with the state of R_i where $d(p) \notin A(L)$, R_i is transformed into the state of R_{i+1}. Note that an RLB L with the state of R_{i+1} can be changed into the state of R_i if a page of L is invalidated

by writing the same page at the other log block. An SLB can be changed into an RLB with the state R_1 or R_2 only if small portion of the SLB is occupied with valid data. While the sequential log block is changed into F state by the partial or switch merge operation, the random log block is changed into F state by the full merge operation. The full merge cost of an RLB with the state of R_i is $N \cdot i \cdot C_{copy} + (i+1) \cdot C_{erase}$.

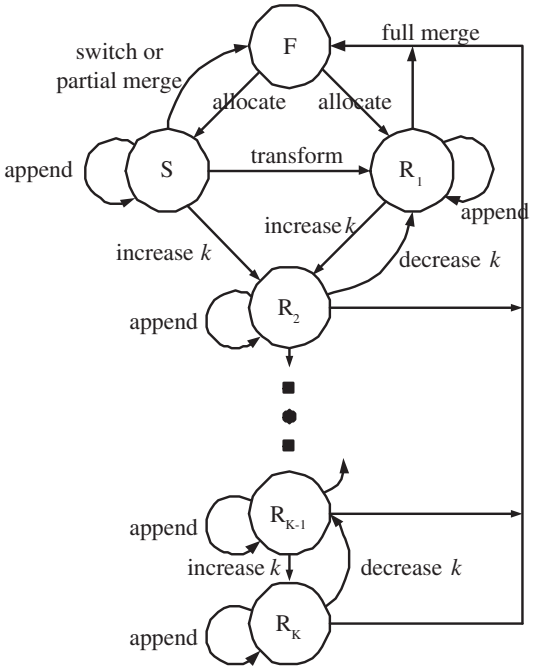


Fig. 3. State Transition Diagram of a Log Block in KAST.

B. Writing in Log Buffer

When a page p should be written at the log buffer, KAST first finds a log block L which has a free space. If there is no log block with a free space, it performs a log block merge. Otherwise, it can write the page p at the log buffer. To write a page in a log block, following steps are required.

The first step is to check whether there is an RLB L such that $d(p) \in A(L)$. If such an RLB exists, we can append the page p to the RLB L without increasing $k(L)$. If the page offset within data block of the page is 0, we can write the page at the first page of an SLB. However, if there is an RLB L such that $d(p) \in A(L)$, we should choose one of two policies as shown in Figure 4. First is to append the page to the RLB without exploiting SLB (Figure 4(b)). Second is to allocate an SLB. For all the pages $p\prime$ where $p\prime \in L$ and $d(p\prime) = d(p)$, we can leave them until there is an update request for the pages (Figure 4(c)) or copy them into the SLB invalidating them in L (Figure 4(d)). In the latter case, to sustain the sequentiality of SLB, we should copy non-updated pages from data block $d(p)$ to SLB (padding).

If there is an SLB S where $d(p) \in A(S)$, we should write the page p into the SLB S. If we can append p without breaking the sequentiality of S, we can write the page p at

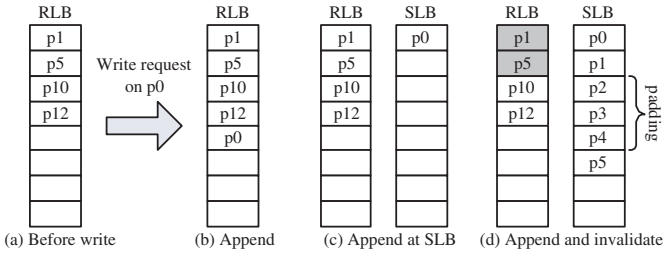


Fig. 4. Writing without increasing associativity.

the SLB S. If there is a small gap between the last written page and the page p, we fill the gap and append the page (Figure 5(b)). However, if there is a large gap or there is a corresponding valid page in the SLB for the page p, we should select one of two policies. First is to perform partial merge for S and write the page p at S during the partial merge (Figure 5(c)). Second is to transform the state of SLB into R_1 by appending the page (Figure 5(d)). Only when the SLB has many free pages, we choose the second policy. In this case, we can expect that there is little possibility of sequential write requests will come for the SLB. In addition, since there are many free pages in the SLB, it is better to use the log block for the on-going write requests instead of performing a partial merge.

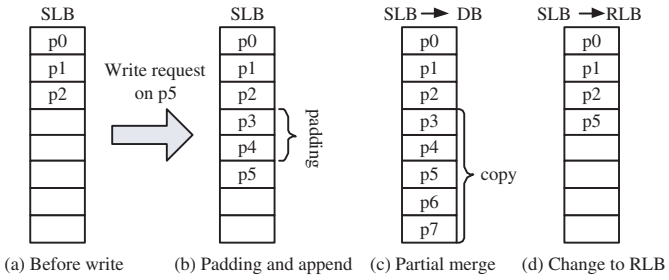


Fig. 5. Transform SLB into RLB.

If we fail to search a log block L such that $d(p) \in A(L)$, we should increase the block associativity of one of log blocks. Before we go to the next step, we check there is an SLB which is complete (i.e., which has no free page). If there exists a complete SLB, we erase the data block associated with the SLB and change the SLB into a data block (i.e., switch merge). Since the switch merge has no page copy overhead, it is better to exploit it than to increase the block associativity of any other log block. If the page offset is 0, we skip the second step and merge one of log blocks to allocate an SLB.

The second step is to select one of log blocks to be transformed, i.e., a victim RLB, to take the page. The RLB which has a lower block associativity and has a larger number of free pages has a higher priority to be selected as a victim. This is to balance the block associativities and the free spaces of log blocks. As long as $k(L)$ of a log block L is below than K, we can increase $k(L)$.

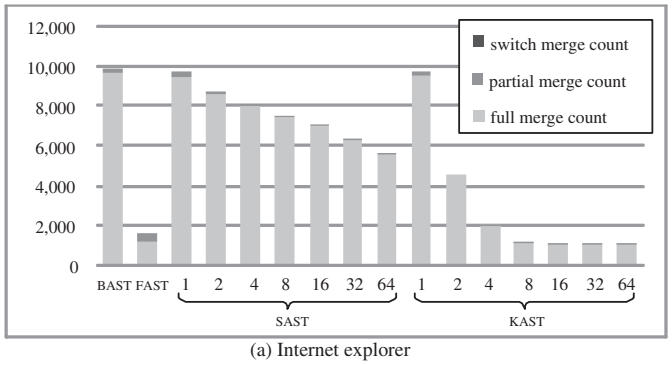
Even though there is an SLB L where $d(p) \notin A(L)$, we

can write the page at the SLB if L has many free pages and is not updated during a long time. Then, the SLB is transformed into an RLB with the state of R_2. If there is no log block which has free space and its associativity is smaller than K, we cannot write a page into one of log blocks.

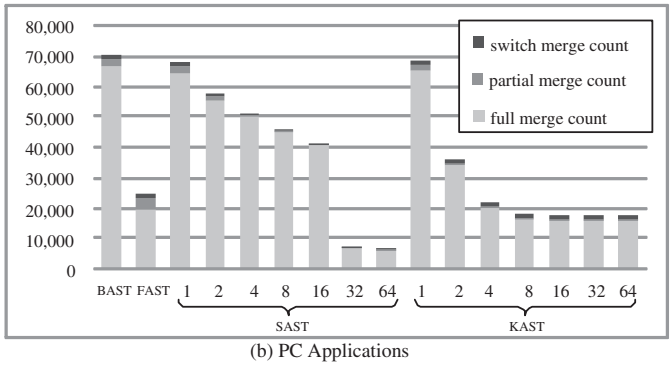
The third step is to merge one of log blocks to make a free space. The first target is an SLB which can be merged by the partial merge. If there are many free pages in the SLBs, we do not perform the partial merge to sustain the SLB. By delaying the partial merge for such an SLB, we can utilize it for the future subsequent write request. The second target is an RLB which has the smallest merge cost and the smallest free pages. The merge cost can be estimated using the log block associativity. By merging one log block, we can get a free log block.

IV. Experiments

We evaluated the performance of KAST scheme using simulation. We used two kinds of I/O workloads for evaluation. One is the storage I/O trace extracted running web browser which generates many small random access files. Another trace is collected running several applications, such as documents editors, music players, web browsers and games in Microsoft Windows XP-based desktop PC. This trace has both sequential access and random access. We used an SLC (single-Level Cell) NAND flash memory model, where a block consists of 64 pages, and the times for page read, page program and block erase are 25μs, 200μs and 2ms, respectively,

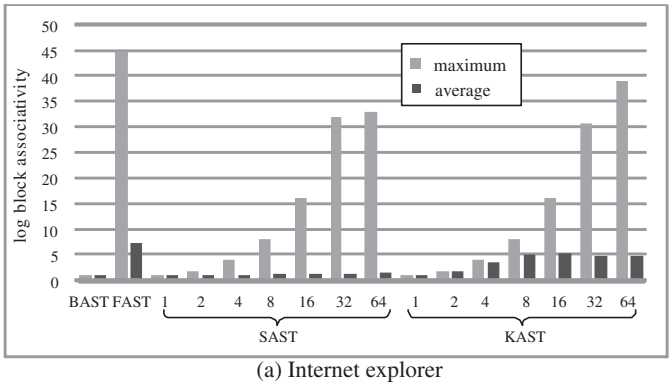


(a) Internet explorer

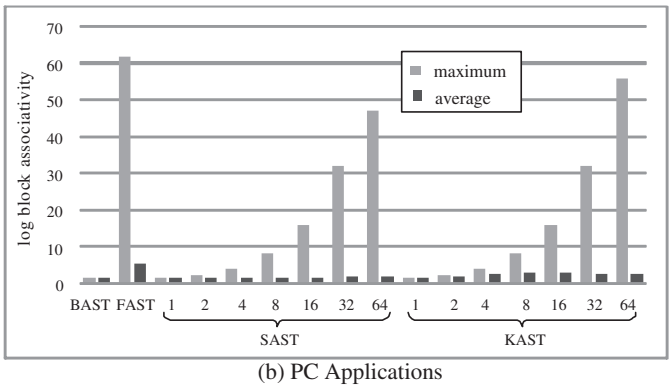


(b) PC Applications

Fig. 6. The number of log block merges.



(a) Internet explorer



(b) PC Applications

Fig. 7. The block associativity.

Figure 6 shows how many number of log block merges are invoked under several FTL schemes. For all experiments, the number of log blocks is 32. For SAST and KAST, we experimented varying the value of K, the maximum value of block associativity of a log block. For SAST, the size of log block set is same to K (i.e., $N = K$). While FAST has a small number of block merges, BAST invokes a significant large number of block merges due to its log block thrashing problem. The numbers of log block merges in SAST and KAST are similar to that of BAST when $K = 1$. But, as K increases, the values are decreased. While the value in KAST is smaller than that of FAST when $K \geq 8$, the value in SAST is larger than that of FAST for most of K values. This is because SAST scheme cannot avoid the log block thrashing problem.

Figure 7 shows the block associativity of log block when a log block is merged. Depending on the maximum association value, the worst-case block merge cost is determined. From the results, we can know that SAST and KAST guarantee the maximum association value is smaller than K. The maximum association value of FAST is significantly high, thus FAST can invoke a long blocking time. But, in SAST and KAST, we can control the blocking time by configuring the value of K. KAST is superior to BAST and SAST for the number of block merges and it is superior to FAST for the cost per block merge (block associativity).

Figure 8 shows the blocking times due to each log block merge. The I/O trace is that of Internet explorer and K is 16

for KAST and SAST schemes. While FAST invokes 548 ms of blocking time at worst case and shows fluctuating costs, KAST requires only 232ms of blocking time. The blocking times of SAST are generally smaller than those of KAST because the log block in SAST is merged before it is filled with data. Therefore, SAST invokes a significant large number of log block merges.

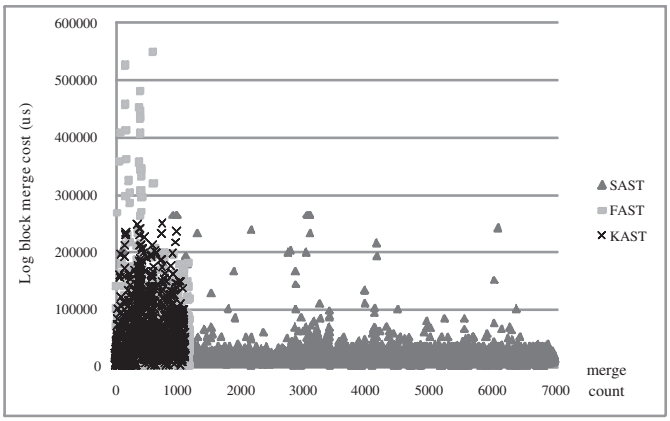
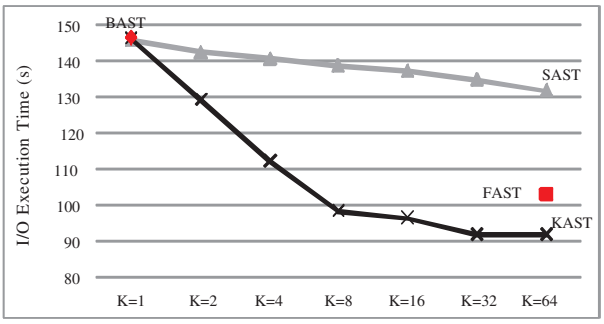
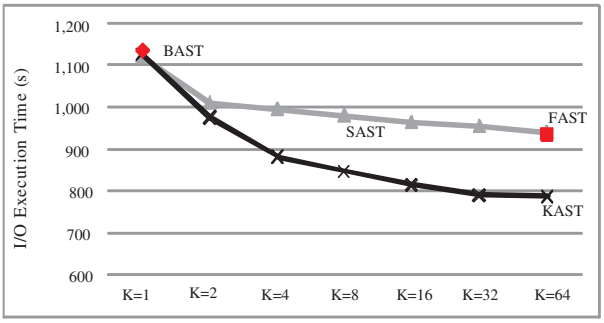


Fig. 8. The I/O execution Time.

Figure 9 shows the total I/O execution time under each FTL scheme. The execution times of SAST and FAST decrease as the value of K increases since the number of log block merges decreases. However, SAST, which cannot avoid the log block thrashing problem, shows a worse result than FAST for the Internet explorer which has random access pattern. However, KAST outperforms both SAST and FAST when $K \geq 8$.



(a) Internet explorer



(b) PC Applications

Fig. 9. The I/O execution Time.

V. Conclusions

In this paper, we proposed a configurable log buffer-based FTL for real-time systems, called KAST, which enables user to configure the worst-case block merge time. In addition, by distributing write requests among log blocks to minimize the log block associativity, KAST shows better average performance than current configurable FTL schemes.

As a future work, we will study a preemptive FTL which can suspend a running block merge operation and execute another write operation with a higher priority.

References

[1] Y. H. Bae. Design of a high performance flash memory-based solid state disk. *Journal of Korean Institute of Information Scientists and Engineers*, 25(6), 2007.

[2] L.-P. Chang, T.-W. Kuo, and S.-W. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems*, 3(4):837–863, 2004.

[3] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A superblock-based flash translation layer for nand flash memory. In *Proc. of International Conference on Embedded Software (EMSOFT)*, pages 161 – 170, 2006.

[4] J.-U. Kang, J. S. Kim, C. Park, H. Park, and J. Lee. A multi-channel architecture for high-performance nand flash-based storage system. *Journal of Systems Architecture*, 53(9):644–658, 2007.

[5] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compact flash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[6] S. Lee, D. Shin, Y. Kim, and J. Kim. Last: locality-aware sector translation for nand flash memory-based storage systems. In *Proc. of IEEE International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED08)*, 2008.

[7] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems*, 6(3), 2007.

[8] S. Y. Park, W. Cheon, Y. Lee, M.-S. Jung, W. Cho, and H. Yoon. A re-configurable ftl (flash translation layer) architecture for nand flash based applications. In *Proc. of International Workshop on Rapid System Prototyping*, pages 202–208, 2007.